C++11 Threads Surprises

Hans-J. Boehm

HP Labs

Outline

- C++11 Threads and Memory model
- Some surprises:
  - Thread cancellation
  - Infinite loops
  - `try_lock()`
  - Detached threads and destructors
- Conclusions

Threads in C++11

- Threads are finally part of the language! (C11, too)
- Threads API
  - Thread creation, synchronization, …
  - Evolved from `Boost.Thread`.
- Memory model
  - Carefully defines shared variable behavior.
    - Still not quite the naïve sequential consistency model.
- Atomic operations
- …

Parallel recursive `fib()` in C++11:

Warning: Incredibly stupid algorithm, but popular example:

```
int fib(int n) {
  if (n <= 1) return n;
  int fib2;
  auto fib1 =
     async([=]{return fib(n-1);});
  fib2 = fib(n-2);
  return fib1.get() + fib2;
}
```

C++11 memory model in a nutshell

- Accessing and modifying same ordinary memory location simultaneously from two different threads is a *data race*.
- Data races are bad: Think
  – Or out-of-bounds array access
  – (Better tools would be nice.)

- Otherwise shared variables behave like you hoped they would
  – Interleave steps from all the threads (seq. consistency)
  – Even better: Sync-free code acts as single step.
- Breaks some common compiler optimizations:
  – Better than breaking user code.

Two common ways to eliminate data races

- Use mutexes:

```
mutex m; int x;
{
  lock_guard<mutex> _(m);
  x++;
}
```

- Use atomics:

```
atomic<int> x;  // data race exempt
x++;
```

Atomics preserve interleaving semantics (by default)

`atomic<int> x,y;` // initially zero

Thread 1	*Thread 2*
`x = 1;`	`y = 1;`
`r1 = y;`	`r2 = x;`

- No data races.
- ***Disallows*** `r1` = `r2` = 0.
- Compiler and hardware do whatever it takes.
  - Usually insert fences, no compiler reordering

Outline

- C++11 Threads and Memory model
- Some surprises:
  - Thread cancellation
  - Infinite loops
  - `try_lock()`
  - Detached threads and destructors
- Conclusions

Standardize existing practice?

- Standards committees sometimes view their charter as standardizing existing tried practice.
  - The C++ committee perhaps a bit less so?
- Nobody should be surprised by the outcome (?)
- Sometimes things don't work out that way.
  - Often, though not always, for good technical reasons

Thread cancellation

- Terminate another thread.
- Posix has `pthread_cancel()`

  incl. dubious asynchronous facilities
- Java has `thread.interrupt()`
  - + dubious asynchronous facilities
- C++11 has

Nothing.

In spite of agreement that we needed something.

Problem: Irreconcilable differences

- Posix:
  - Cancellation is not ignorable.
  - There is no way to return to mainline code once a thread is cancelled.
    - and that's viewed as critically important.
  - Correct code typically uses `pthread_cleanup…`
- C++:
  - Existing cleanup mechanism: Exceptions.
  - Code is written to deal with exceptions, not pthread_cleanup…
  - No practical way to prevent swallowing exception.

Outline

- C++11 Threads and Memory model
- Some surprises:
  - Thread cancellation
  - Infinite loops
  - `try_lock()`
  - Detached threads and destructors
- Conclusions

- Consider:

Thread 1

```
for (i = 0; i < 10; i += n) {x++;}
for (i = 0; i < 10; i += n) {y++;}
```

Thread 2

```
r = y;
```

- Data race with n = 1? Yes.
- Data race with n = 0? No.

- After loop fusion:

Thread 1

```
for (i = 0; i < 10; i += n) {x++; y++;}
```

Thread 2

```
r = y;
```

- Data race with n = 1? Yes.
- Data race with n = 0? Yes!

Options

- Outlaw transformations like loop fusion on potentially infinite loops.
  - Likely to hurt important optimizations.
  - Clean semantics.
  - Java follows this route.
- Allow transformation.
  - Messy spec? Complicated programming rules?
  - Allows optimizations.

Deciding factor:

- Existing practice:
  - Many compilers eliminate "dead" loops, even if they're infinite.
    - See John Regehr's (later) blog "Compilers and termination revisited".
  - Already really hard to say what infinite loops mean.

C++11 "Solution"

- "The implementation may assume that any thread will eventually do one of the following:
  - terminate,
  - make a call to a library I/O function,
  - access or modify a volatile object, or
  - perform a synchronization operation or an atomic operation."
- *Effectively outlaws side-effect-free and sync-free infinite loops.*
- Allows loop optimizations.
- Provides a way to write infinite loops.
- Doesn't break currently portable code.

Outline

- C++11 Threads and Memory model
- Some surprises:
  - Thread cancellation
  - Infinite loops
  - `try_lock()`
  - Detached threads and destructors
- Conclusions

try_lock()

Consider:

```
int x; mutex m;
```

Thread 1

```
x = 42;
m.lock();
```

Thread 2

```
while(m.try_lock())
        m.unlock();
assert(x == 42);
```

Can the assertion fail?

In real implementations: Yes.

Thread 1 statements can be reordered.

Preventing this can be expensive. Affects `m.lock()` impl.

C++11 treatment of `trylock()`

- `try_lock()` can spuriously fail to acquire mutex.
  - even when mutex was never held.
  - Equivalently: System can acquire mutex.
- Implementations shouldn't really do that!
- But `try_lock()` failure ➔ nothing!
- code that could detect reordering now has data race.

Thread 1

```
x = 42;
m.lock();
```

data race

Thread 2

```
while(m.try_lock())
        m.unlock();
assert(x == 42);
```

Outline

- C++11 Threads and Memory model
- Some surprises:
  - Thread cancellation
  - Infinite loops
  - `try_lock()`
  - Detached threads and destructors
- Conclusions

“Detached” threads

- Threads that can no longer be “joined” (waited for).
- Posix allows detached threads.
- Boost threads allowed detached threads.
  - Destroying an unjoined thread implicitly detaches.
  - Seems natural enough, but …

An implicit detach problem:

```
int fib(int n) {
  if (n <= 1) return n;
  int fib1, fib2;
  thread t([=, &fib1]{fib1 = fib(n-1);});
* fib2 = fib(n-2);
  t.join();
  return fib1 + fib2;
}
```

What if an exception is thrown at * ?

1. Call to `t.join()` is not executed.
2. Thread `t` is destroyed ➔ detached.
3. Child is still running, writes to local `fib1` in *parent* thread.
4. Undebuggable crash.

Complication: Emulating join is hard

```
thread_local T x;  atomic<bool> t2done;
```

Main thread:

```
create thread 2;

while (!t2done) {}
```

return from main;

destroy T's allocator;

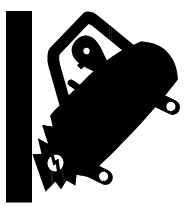
Thread 2:

```
…; x = …;
t2done = true;
```

Destroy thread 2's `x`

Also important to wait for destruction of `thread_local`s!

Which might be introduced by libraries you can't see.



C++11 treatment

- Some support for detached threads:
  - `detach()`
  - `quick_exit()`
  - `notify_all_at_thread_exit()`
- Recommendation: Just call `join()`!
- No implicit detach!
- Destruction of unjoined thread invokes `terminate()`!

Outline

- C++11 Threads and Memory model
- Some surprises:
  - Thread cancellation
  - Infinite loops
  - `try_lock()`
  - Detached threads and destructors
- Conclusions

Some surprises, usually for good reasons

- No thread cancellation:
  - Somewhat political issue, but
  - No fully compatible forward path.
- *Undefined infinite loops:
  - Really preserves status quo.
  - Which already surprises people.

- *Disallow common optimizations:
- *Spurious `try_lock()` failures:
- No implicit detach:
  - Traditional approaches are inherently brittle (or worse).
  - C++11 allows robust solutions.

* also in C11

Questions?

Memory model references

- Boehm, Adve, You Don't Know Jack About Shared Variables or Memory Models , Communications of the ACM, Feb 2012.
- Boehm, “Threads Basics”, HPL TR 2009-259.
- Adve, Boehm, “Memory Models: A Case for Rethinking Parallel Languages and Hardware, Communications of the ACM, August 2010.
- Boehm, Adve, “Foundations of the C++ Concurrency Memory Model”, PLDI 08.
- Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber, “Mathematizing C++ Concurrency”, POPL 2011.

Easily understandable

Mathematically rigorous

C++ specific